NIUF 411-92

North American ISDN Users' Forum Application Software Interface (ASI)

Part 4: UNIX Access Method (Version 1.0)

Approved: October 30, 1992

Application Software Interface Expert Working Group
ISDN Implementor's Workshop
North American ISDN Users Forum

Revision History

October 1992	Baseline Approved Document (NIUF 411-92)
February 1993	Editorial corrections

Abstract

This document describes the Application Software Interface (ASI) access method for the UNIX operating system. Although it is a desired goal to have a common ASI for all operating systems, this specification will only apply to the UNIX operating system environments. The following restrictions will also apply:

- The access method uses UNIX System V STREAMS (or compatible STREAMS implementations such as OSF/1). Only release 3.2 STREAMS features are utilized.
- In the context of UNIX, a Program Entity (PE) is a process. A single ASI Entity (AE) can associate with only a single PE. A PE may associate with multiple AEs.
- The UNIX access method identifies a point over which user plane data is transferred to/from an ISDN connection. It does not at this time attempt to define the contents or format of the user plane connection beyond specifying that it will use STREAMS.

Keywords

application programming interface; API; Application Software Interface; ASI; access method; implementation agreement; integrated services digital network; ISDN.

Notice of Disclaimer

This specification was developed and approved by organizations participating in the North American ISDN Users' Forum (NIUF) meetings in October 1992. The National Institute of Standards and Technology (NIST) makes no representation or warranty, express or implied with respect to the sufficiency, accuracy, or use of any information or opinion contained herein. The use of this information or opinion is at the risk of the user. Under no circumstances shall NIST be liable for any damage or injury incurred by any person arising out of the sufficiency, accuracy, or use of any information or opinion contained herein.

Acknowledgments

NIST would like to acknowledge the NIUF Application Software Interface Expert Working Group, and especially the following individuals, for their valuable contributions to this document:

Kenneth A. Argo	Fujitsu America, Inc.
Ron Bhanukitsiri	Digital Equipment Corp.
Cheng T. Chen	Teleos Communications, Inc.
Stephen Halpern	NYNEX Science and Technology
Frank Heath	Rockwell CMC
Stephen Rogers	Electronic Data Systems
Chris Schmandt	MIT Media Lab
Ben Stoltz	Sun Microsystems, Inc.
Robert E. Toense	NIST
Adrian Viego	Bellcore
Wayne Yamamoto	Sun Microsystems, Inc.

1.0. Scope

This document describes the Application Software Interface (ASI) access method for the UNIX operating system. This access method specification will only apply to the UNIX operating system environments. The following restrictions will also apply:

- The access method uses UNIX System V STREAMS (or compatible STREAMS implementations such as OSF/1). Only release 3.2 STREAMS features are utilized.
- In the context of UNIX, a Program Entity (PE) is a process. A single ASI Entity (AE) can associate with only a single PE. A PE may associate with multiple AEs.
- The UNIX access method identifies a point over which user plane data is transferred to/from an ISDN connection. It does not at this time attempt to define the contents or format of the user plane connection beyond specifying that it will use STREAMS.

2.0. Introduction

The B reference is defined in section 2.3 of the ASI Overview and Protocols document.This document deals with the problem of creating an association between multiple entities below the B reference point (called ASI Entities (AEs)) and a single entity (called the Program Entity (PE)). Issues addressed here are the following:

- Determining the number and names of AEs;
- Carrying out the necessary hardware initialization of the interfaces, including downloading code if necessary;
- Pushing (i.e., I_PUSH) any interface specific STREAMS modules that are required (if any);
- Linking (i.e., I_LINK) any interface specific STREAMS multiplexors;
- Starting any user space processes (daemons) that may be necessary.

When the PE has completed the above, the AE is available for use.

Once the interfaces are up, the Server would use STREAMS getmsg/putmsg system calls to pass ASI messages across the B reference point (See appendix A).

3.0. Overview

The use of STREAMS as the basis for ASI access satisfies the bidirectional asynchronous operations of the ASI. In addition, STREAMS provides a general, flexible facility for communication services. The principal facilities provided by STREAMS are:

- Buffer management,
- Flow control,
- Scheduling,
- Multiplexing,
- Asynchronous operation of STREAMS and user processes,
- Error and trace logging.

The UNIX access method utilizes the UNIX system calls specified in Appendix B to access the ASI. ASI Reference Point B commands from Part 1 are always passed in the “data” portion of putmsg or getmsg function calls.

ASI functionality requires that the ASI must provide three basic services: management, control, and data. The management and control services, in the UNIX access method, are implemented as a single ASI function. There can be only one management/control (M/C) function per AE. This is also true for the PE. The access method defines a means to establish multiple user plane connections.

When the AE's service is needed, the PE will format a message, as defined in Part 1 Overview and Protocol. This message will be passed across the interface to the AS's Stream head by calling the UNIX putmsg() system call. Likewise, the AE passes an ASI message to the PE by queueing the message to the PE's Stream head; the PE then retrieves the message by calling the UNIX getmsg() system call. When the PE no longer requires the service from the AE, the PE closes the Stream, via the UNIX close() system call.

4.0. Initialization and Binding

Before the access method can be used, it must be initialized. While there are many possible implementations of AEs, the manner in which a PE initializes an AE must be consistent across all implementations to assure portability.

The process of initialization and binding consists of three phases: AE Preparation, PE-AE Stream Head Open, and AE Start. Some implementations may have no action performed in some of these phases. The phases are sequential; a failure return code (i.e., non-zero) will abort the subsequent phases and result in the interface being unavailable for use.

4.1. Initialization and Binding Phase Definition

The process of initialization and binding supports a variety of implementation architectures, allowing the AE to be implemented as an add-in board, as software in the kernel, or as a user process. The three steps of initialization and binding enable this range of configurations.

4.1.1. AE Preparation

AE preparation is low level initialization before an AE can be instantiated. It includes, for example, downloading of code and starting adapter hardware.

A vendor-supplied program/script carries out the actual work of initialization. The PE is supplied with the name of the script. Calling the script prepares exactly one AE and returns success or failure. A key may be passed to the script so it can deal with the proper AE, due to the fact that multiple AEs may be present. Each interface may be brought up or marked "down" individually.

This phase may log progress, and any errors for later examination. The PE receives a simple success or failure indication.

4.1.2. PE-AE Stream Head Open

In this phase, the PE opens the Stream associated with the AE, using the "open" system call.

At this point the PE can safely attempt an open("/dev/Fubar_Inc_Asi0",O_RDWR) call.

4.1.3. AE Start

This phase carries out the STREAMS manipulation operations via a vendor-supplied program. At this point, the PE has the file descriptor of the Stream that will be used as the access path for ASI messages to the AE. This Stream may require further manipulation to be fully usable. It is envisioned that modules would be pushed, and multiplexors linked.

The vendor supplied program's Stream manipulation has to be done on the PE-AE Stream already in existence, and must remain in effect when the vendor program exits. For this reason the file descriptor must be passed to the vendor program in its open file descriptor table, i.e., via a UNIX fork/exec sequence.

4.2. Initialization and Binding Operation Mechanisms

This section describes details for configuring the three phases described in the previous section.

4.2.1. Master Configuration File Format

There is a master configuration file that is used by the PE for initialization of the AE. It is supplied to the PE when the PE is started, i.e.,

```
PE_run /etc/config_master.normal
```

In this example, “PE_run” is a command to start a PE and “`/etc/config_master.normal`” is the argument indicating the location of the master configuration file. The master configuration file is used by the PE during initialization of the AEs. This file contains an entry for each AE.

Each entry in this master configuration file consists of a single line terminated by a newline ('\n'). The newline at the end of the file is optional. Each line begins with a tag followed by a colon, if there are parameters for that tag. The fields for an entry are delimited by white space. Lines beginning with '#' or ‘\n’ (newline) are ignored.

All AEs have an entry in the PE’s master configuration file.

For each AE, there is a line in the master configuration file with the tag “ASI_NAME”. An ASI_NAME entry has two white space delimited parameters. The first is a symbolic name for the AE. The second is a pathname for the configuration file for the ASI.

```
ASI_NAME: <symbolic name> <pathname of this ASI’s configuration file>
```

Note: This defines one line in the master configuration file.

Examples:

```
ASI_NAME: rboc /etc/phone/asi/rboc.config
ASI_NAME: test-line /etc/phone/asi/vendor_a.config
ASI_NAME: isdn1 /etc/phone/asi/att5e8.212.555.1212
ASI_NAME: corp-pbx /etc/phone/asi/corpnet.config
ASI_NAME: my_isdn /etc/phone/asi/my.config
```

4.2.2. AE Configuration File Format

Each AE’s individual configuration file contains entries that are used by the PE. Commands are executed using the UNIX system calls fork() and exec().

Each entry in this AE configuration file consists of a single line terminated by a newline ('\n'). The newline at the end of the file is optional. Each line begins with a tag followed by a colon, if there are parameters for that tag. The fields for an entry are delimited by white space. Lines beginning with '#' or ‘\n’ (newline) are ignored.

No file name expansion or I/O redirection is done on the commands or arguments before the exec. The ‘\’ character will escape white space in the “execv” argument lists. Sample entries in the ASI configuration file are:

```
INIT.AE_PREP: <command 1> <arg1> … <argn>
INIT.AE_OPEN: <pathname that will eventually supply ASI semantics>
INIT.AE_START: <command 2> <arg1> … <argn>
```

Null phases may be denoted by omitting the entry for that phase or by terminating the line after the colon following the tag for that phase (i.e., no command or parameter list).

4.2.2.1. AE Configuration File Entries and Functions

INIT.AE_AUTOSTART (optional)

If this parameter is present and the value is “TRUE,” then the PE will start this AE at PE initialization time. Otherwise, the AE will not be started at initialization time.

The environment variable “ASI_NAME” contains the symbolic name of the AE found in the master configuration file. This tag can be used to locate other configuration information that may be required for the initialization or control of the particular AE. The environment variable “ASI_MASTER,” the name of the master configuration file, is also inherited.

INIT.AE_PREP (optional)

The entry tagged INIT.AE_PREP initiates the AE Preparation phase.

The entry is composed of the key “INIT.AE_PREP.KEY” followed by at least one tab or space character followed by a value which is a string of characters. The value string is composed of characters not including spaces, tabs,

newlines, or nulls. The text line is terminated by a null character. The following regular expression formally defines the text line.

```
"INIT.AE_PREP.KEY" [ \t]+ [^ \t\n\0]+ \n
```

INIT.AE_PREP is for board level initialization and setting up the AE. The first argument following the key is the command that the PE forks and then execs. Subsequent arguments are arguments to the command forked and execed.

INIT.AE_PREP may create processes, or precondition the device named by INIT.AE_OPEN so the subsequent open will find the correct device or pseudo-device, or process.

The INIT.AE_PREP command is for the first phase of initialization, i.e., board level initialization and setting up the AE. The INIT.AE_PREP command is an optional command; the board may need no initialization.

The AE_PREP process emits a tag and string on its standard output (stdout). The maximum number of characters from start of tag to '\n' inclusive is 1024 characters.

Note: The PE implementation should make provisions for multiple lines of text to be emitted by the AE_PREP process' standard output.

The environment variable "ASI_NAME" contains the symbolic name of the AE found in the master configuration file. This tag can be used to look up other configuration information that may be required for the initialization or control of the particular AE. The environment variable "ASI_MASTER," the name of the master configuration file, is also inherited.

INIT.AE_OPEN (mandatory)

The INIT.AE_OPEN entry contains the pathname to the ASI Stream device head. The filename provided is opened. 'B' reference point messages are carried across this file descriptor. This may additionally require completion of the INIT.AE_START and INIT.AE_CONTROL phases.

The entry is composed of the key "INIT.AE_OPEN.KEY" followed by at least one tab or space character followed by a value which is a string of characters. The value string is composed of characters not including spaces, tabs, newlines, or nulls. The text line is terminated by a null character. The following regular expression formally defines the text line.

```
"INIT.AE_OPEN.KEY" [ \t]+ [^ \t\n\0]+ \n
```

The environment variable "ASI_NAME" contains the symbolic name of the AE found in the master configuration file. This tag can be used to locate other configuration information that may be required for the initialization or control of the particular AE. The environment variable "ASI_MASTER," the name of the master configuration file, is also inherited.

INIT.AE_START (optional)

INIT.AE_START performs the AE Start phase.

INIT.AE_START is an optional command. INIT.AE_START is for further configuration of the AE Stream after the successful completion of INIT.AE_OPEN. The INIT.AE_START command is executed by the PE. INIT.AE_START expects to find the file descriptor for the AE Stream on file descriptor 3 (the file descriptor from the open(INIT.AE_OPEN)).

The AE_START process emits a tag and string on its standard output (stdout). The maximum number of characters from start of tag to '\n' inclusive is 1024 characters.

Note: The PE implementation should make provisions for multiple lines of text to be emitted on the AE_START process' standard output. The text line is composed of the key "INIT.AE_START.KEY" followed by at least one tab or space character followed by a value which is a string of characters. The value string is composed of characters not including spaces, tabs, newlines, or nulls. The text line is terminated by a null character. The following regular expression formally defines the text line.

```
"INIT.AE_START.KEY" [ \t]+ [^ \t\n\0]+ \n
```

The environment variable “ASI_NAME” contains the symbolic name of the AE as identified in the master configuration file. This tag can be used to locate other configuration information that may be required for the initialization or control of the particular AE. The environment variable “ASI_MASTER,” the name of the master configuration file, is also inherited.

The values returned from the commands should be treated as privileged information so that security mechanisms that use this information can be implemented properly.

STREAMS modules may be PUSHED onto the file descriptor, or STREAMS multiplexors may be linked under the Stream.

The environment variable “ASI_NAME” contains the symbolic name of the AE found in the master configuration file. This tag can be used to locate other configuration information that may be required for the initialization or control of the particular AE. The environment variable “ASI_MASTER,” the name of the master configuration file, is also inherited.

INIT.AE_CONTROL (optional)

If this parameter is present, then messages are sent on the AE file descriptor using information from INIT.AE_PREP and INIT.AE_START. The messages are sent using putmsg() on the AE file descriptor (see the code example below). The values returned from the commands should be treated as privileged information so that security mechanisms that use this information can be implemented properly.

STREAMS modules may be PUSHED onto the file descriptor, or STREAMS multiplexors may be linked under the Stream.

The environment variable “ASI_NAME” contains the symbolic name of the AE found in the master configuration file. This tag can be used to locate other configuration information that may be required for the initialization or control of the particular AE. The environment variable “ASI_MASTER,” the name of the master configuration file, is also inherited.

Code example:

```
/*
 * ASI reference point B UNIX access method. “Control”
 */
#define UAM_PROTOCOL   (1)    /* Ref. Pt. B UNIX Access Meth. protocol ID */

/* ASI Reference Point B, UNIX access method message definitions */
typedef enum {
       INIT_AE_NOOP = 0,
       INIT_AE_CONTROL_REQ,
       INIT_AE_CONTROL_CNF,
} asibuam_msg_t;

/* Definitions for INIT_AE_CONTROL messages */
#define INIT_AE_MAX_STRING (1024)
#define INIT_AE_PREP_KEY (1)
#define INIT_AE_START_KEY (2)

typedef struct {
       int     protocol;
       int     message_type;
       int     key;
       char    value[INIT_AE_MAX_STRING];
} ae_uam_msg_t;

#define AE_CONTROL_FAIL (0)
#define AE_CONTROL_SENT (1)
#define AE_CONTROL_OK (2)

/* All of the above definitions are normally from a header file */

ae_control(fd, ae_prep_string, ae_start_string)
       int     fd;
       char    *ae_prep_string;
       char    *ae_start_string;
{
```

```
        if (ae_prep_string != NULL) {
                if (ae_control_msg(fd, INIT_AE_START_KEY, ae_prep_string)
                 == AE_CONTROL_FAIL) {
                        return (AE_CONTROL_FAIL);
                }
        }

        if (ae_start_string != NULL) {
                if (ae_control_msg(fd, INIT_AE_START_KEY, ae_start_string)
                 == AE_CONTROL_FAIL) {
                        return (AE_CONTROL_FAIL);
                }
        }
        return (AE_CONTROL_OK);
}

int
ae_control_msg(fd, key, string)
        int     fd;
        int     key;
        char    *string;
{
        struct strbuf ctl;
        ae_uam_msg_t control_msg;

        control_msg.protocol = UAM_PROTOCOL;
        control_msg.type = INIT_AE_CONTROL_REQ;
        control_msg.key = key;
        strcpy(control_msg.value, string);
        ctl.maxlen = sizeof(control_msg);
        ctl.len = sizeof(control_msg);
        ctl.buf = (char *)&control_msg;
        i = putmsg(fd, &ctl, NULL, 0);
        if (i < 0) {
                perror("ae_control_msg: putmsg failed");
                return (AE_CONTROL_FAIL);
        }

        {
                struct strbuf  ctl;
                ae_uam_msg_t   control_msg;
                int                   flags;

                ctl.maxlen = sizeof(control_msg);
                ctl.buf = (char *)&control_msg;
                flags = 0;
                if (getmsg(fd, &ctl, NULL, &flags) < 0) {
                        perror("ae_control: getmsg");
                        return (AE_CONTROL_FAIL);
                }
                if (control_msg.protocol != UAM_PROTOCOL) {
                        return (AE_CONTROL_FAIL);
                }
                if (control_msg.type != AE_INIT_CONTROL_CNF)
                        return (AE_CONTROL_FAIL);
        }
        return (AE_CONTROL_OK);
}
```

SHUTDOWN.AE_STOP (optional)

SHUTDOWN.AE_STOP is a UNIX command to cleanly shutdown the AE. The PE passes the AE file descriptor as this command's file descriptor #3. The exit code of this command is always zero. After this command exits, the PE closes the AE file descriptor.

SHUTDOWN.AE_KILL (optional)

If open, the PE closes the AE file descriptor and then executes the command. The exit code of this command is always zero.

Note that if SHUTDOWN.AE_STOP and SHUTDOWN.AE_KILL are not provided, then it will be assumed by the PE that a simple close of the AE file descriptor is sufficient to shut down the AE.

The environment variable “ASI_NAME” contains the symbolic name of the AE found in the master configuration file. This tag can be used to locate other configuration information that may be required for the initialization or control of the particular AE. The environment variable “ASI_MASTER,” the name of the master configuration file, is also inherited.

Sample configuration file:

```
INIT.AE_PREP: /etc/asi_card_1_ae_prep -log /dev/null foo
INIT.AE_OPEN: /dev/asi_card_1_control
INIT.AE_START: /etc/asi_card_1_ae_start -log /dev/null 23
```

4.3. Implementation Examples

To illustrate the necessary steps needed to bring up the PE/AE association, three implementation models, which highlight different portions of the initialization problem, are given below. They represent only three choices on the spectrum of designs. The intent is to illustrate possible choices, not to dictate implementation.

4.3.1. Implementation I

Implementation I makes use of a very “smart” card which has all of its intelligence on board, that is, all of the AE’s functionality is across the bus. It has only a STREAMS driver provided by the vendor and the system-supplied Stream head.

This interface is virtually all AE Preparation, as follows:

1. AE Preparation: The board would have to be reset, its basic functionality checked, and then code would probably have to be downloaded. No daemons started.
2. PE-AE Stream Head Open: Open the device.
3. AE Start: Null.

4.3.2. Implementation II

Interface II is a “dumb” card with a STREAMS driver and one or more STREAMS modules that provide the AE’s functionality.

In this case, most initialization is carried out by AE_Start.

1. AE Preparation: Any system initialization necessary which may include: setting up chips, memory mapping, etc.
2. PE-AE Stream Head Open: Open the device.
3. AE Start: Push and Link all the modules.

4.3.3. Implementation III

Interface III is again a “dumb” card with only HDLC/physical layer functionality on board and with the rest of the ASI functionality in a user-space program. The interface between the board and the user-space program need not be STREAMS. This interface is internal to the AE. To provide a STREAMS interface to the Server (PE) a vendor supplied pseudo device is provided which links Server and Daemon together.

1. AE Preparation: Any system initialization necessary which may include: setting up chips, memory mapping, etc.
2. PE-AE Stream Head Open: Open the PE side of the STREAMS pseudo device.

3. AE Start: Start up daemon, open whatever internal drivers necessary to talk to the devices. Open the AE side of the STREAMS pseudo device.

If an implementation mixes Interface II and Interface III, using STREAMS modules and a User Space presence, the PE does not push the modules. The User space program owns the Stream and does its own pushing. The external interface is still the STREAMS pseudo device.

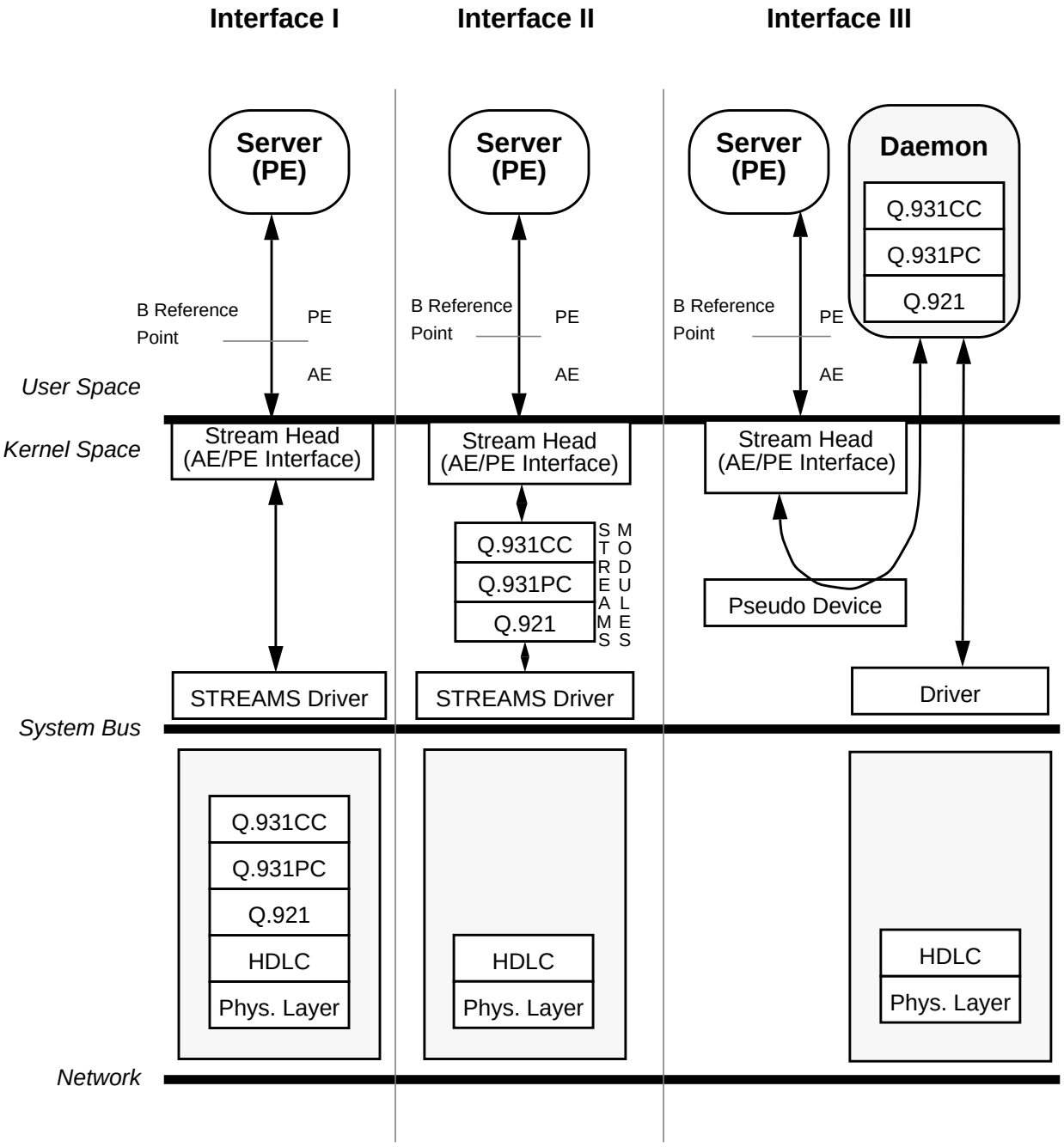


Figure 1: Implementation Strategies for ASI Call Control.

5.0. Access to Data Channels

Data channels are accessed by the PE through the UNIX file system. A UNIX STREAMS device (file) is associated with each data channel. To read or write data to the data channel, the PE opens the appropriate device. Because the device is a UNIX Stream, putmsg() can be used to write data to the channel and getmsg() can be used to read data from the channel.

When an AE is installed, a UNIX device shall be created for each channel available from the AE.

The mapping of a data channel to a UNIX device name is handled though the AE's configuration file. The configuration file for the AE will contain an entry for each channel. The tag for each entry will be "CHANNEL." The CHANNEL entry will have three arguments; channel_type, channel_number, and device name (Note: Channel_type and channel_name are defined in "Part 1: Overview and Protocols" and are used to identify a channel. Device name is a UNIX device).

An example CHANNEL entry is:

```
CHANNEL: 1 2 /dev/isdn/card1b1
```

The channel_type is 1 (indicating that the channel is a B channel) and the channel number is 2. Hence, data associated with B2 for the AE is available by opening the device "/dev/isdn/card1b1."

Channels are identified through the "Channel" constructed data type defined in section 6.2.2 of Part 1, Overview and Protocols, (Version 1.0). Channels can also be identified as logical channel types, the enumeration of which is for future study. To enhance the readability of the configuration file, the following strings are synonyms for the channel_types defined in "Part 1: Overview and Protocols" of the ASI specification.

```
# Type      Channel    Name
D_CHANNEL   0x0        # A D channel
B_CHANNEL   0x1        # A B channel
H0_CHANNEL  0x6        # An H0 channel
H10_CHANNEL 0x17       # An H10 channel
H11_CHANNEL 0x18       # An H11 channel
```

Data can be read or written by the PE, once the channel device has been opened. Data is in the proper format for the service provided (e.g., μ-law audio or HDLC data).

Appendix A: Quick Introduction to STREAMS

The mechanism in which the ASI access method is implemented is chosen to be the UNIX System V STREAMS or compatible UNIX System V STREAMS (e.g., from OSF/1).

STREAMS is an asynchronous message passing mechanism between a service user and service provider. The service user utilizes the following UNIX system calls to manipulate the Stream head:

- open()
- fcntl()
- ioctl()
- putmsg()
- getmsg()
- I_LINK ioctl()
- I_PUSH ioctl()
- poll()
- close()

To access STREAMS, the service user opens a Stream to the service provider, via the UNIX open() system call. A unique STREAMS identifier is returned by the service provider to the service user in the form of a UNIX file descriptor. The service user then passes a message to the service provider by calling the UNIX putmsg() system call. Likewise, the service provider passes a message to the service user by queueing the message to the service user's Stream head. The service user then retrieves the message by calling the UNIX getmsg() system call. The service user terminates the service "agreement" by closing down the STREAMS, via the UNIX close() system call.

For further information on STREAMS, please refer to the INTRODUCTION TO UNIX STREAMS or UNIX SYSTEM V RELEASE 3.2 Programmer's Guide: STREAMS manual.

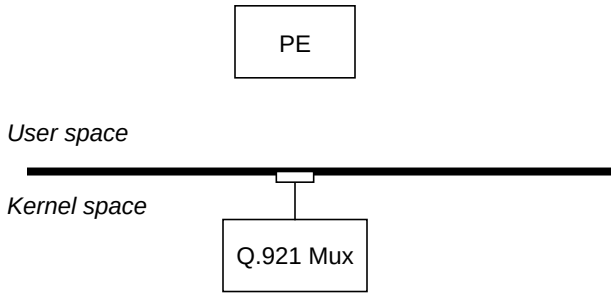
Appendix B: Example of Passing ASI Messages

```
/*
 * ASI Reference point B commands from Part 1 are always passed in
 * the “data” portion of putmsg or getmsg.
 */

#define INTERFACE_ASI   (0x10)
#define PLANE_CONTROL   (0x02)
#define ASI_DON_T_CARE  (0x0)
#define Nb_CONNECT_req  (0x103)

originate_cmd(ae_fd)
       int ae_fd;
{
       int i, j,flags = 0;
       struct strbuf    data;
       ASI_Cmd cmd, test_cmd;

       data.maxlen = 800;
       data.len = sizeof(cmd);
       data.buf =(char *) &cmd;

       cmd.PI   = INTERFACE_ASI | PLANE_CONTROL;
       cmd.AEI  = ASI_DON_T_CARE;
       cmd.PEI  = 23;/*Made up value*/
       cmd.CMD  = NB_CONNECT_req;
       cmd.LEN  = 0; /*Will be filled in later*/
       /* Put in some ASI command and parameters here*/

       i = putmsg(ae_fd, NULL, &data, 0);
       return 0;
}
```

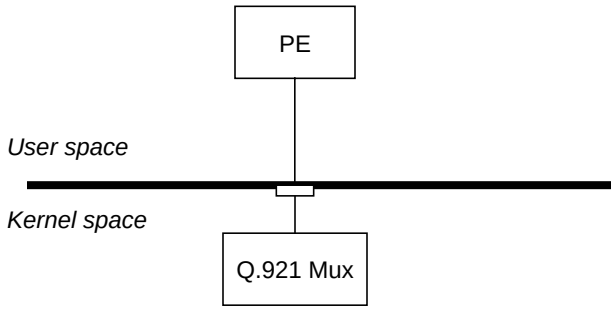
Appendix C: Illustrated Example of Initialization and Binding Process

This process illustrates the initialization and binding process as may be performed with an Implementation II configuration as described in 4.3.2.

1. PE prior to initialization and binding.

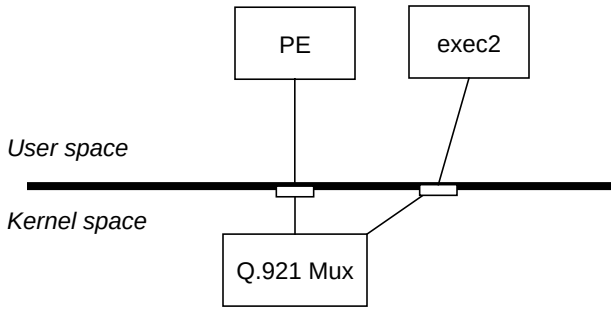


2. PE opens AE/PE Stream head as defined by the INIT.AE_OPEN entry in the configuration file.

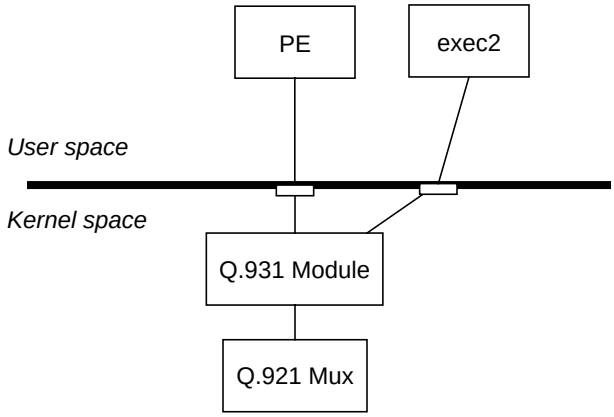


The following steps would be performed by the process defined by the INIT.AE_START entry in the configuration file.

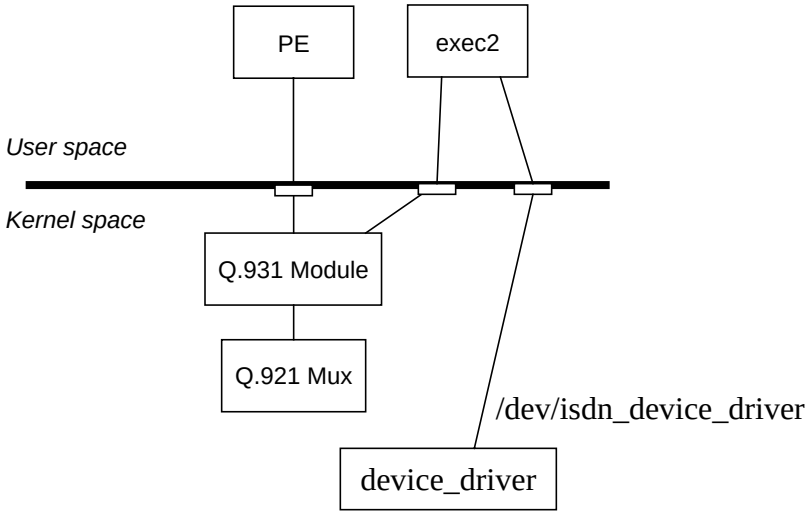
3. PE performs “fork” and “exec” of exec2. Exec2 is a vendor-supplied process that will construct the appropriate “stack” of STREAMS modules for the PE.



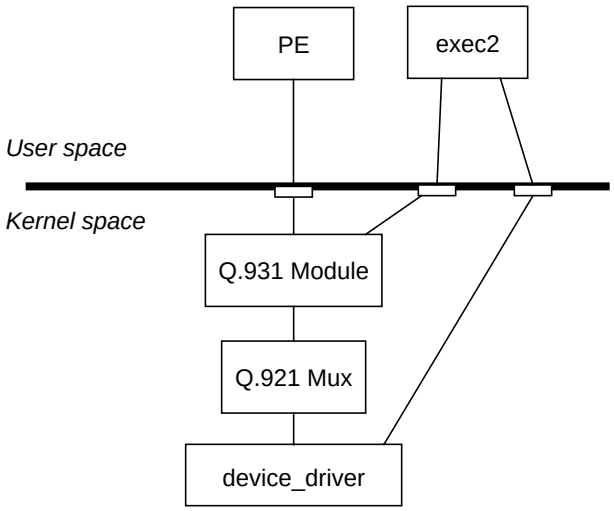
4. Exec2 pushes a Q.931 STREAMS module.



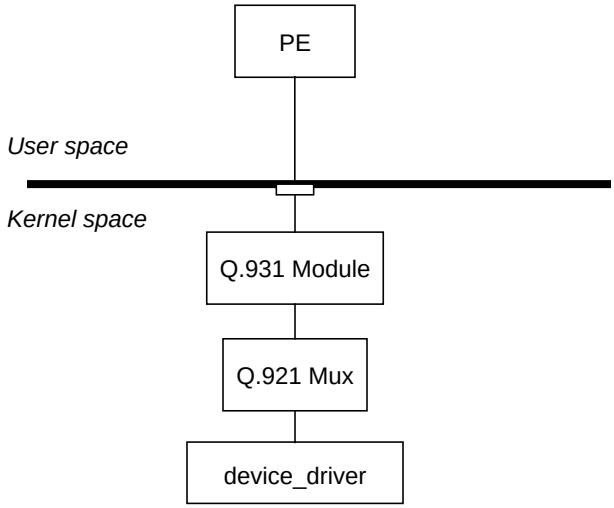
5. Exec2 opens “/dev/isdn_device_driver.”



6. Exec2 links “device_driver” under Q.921 multiplexor.



7. Exec2 exits AE leaving PE initialized and bound.



Appendix D: References

D.1 ANS Documents

[1] ANS T1.607-1990, *Telecommunications — Integrated Services Digital Network (ISDN) — Digital Subscriber Signalling System Number 1 (DSS1) — Layer 3 Signalling Specification for Circuit-Switched Bearer Service*.

D.2 CCITT Documents

[2] CCITT Recommendation I.320 - 1988, *ISDN Protocol Reference Model*.

[3] CCITT Recommendation I.515 - 1988, *Parameter Exchange for ISDN Networking*.

[4] CCITT Recommendation Q.921-1988 (also designated CCITT Recommendation I.441-1988), *ISDN User-Network Interface Data Link Layer Specification*.

[5] CCITT Recommendation Q.931-1988 (also designated CCITT Recommendation I.451-1988), *ISDN User-Network Interface — Layer 3 Specification for Basic Call Control*.

[6] CCITT Recommendation V.110 -1988, *Support of Data Terminal Equipments (DTEs) with V-series Type Interfaces by an Integrated Services Digital Network (ISDN)*.

[7] CCITT Recommendation V.120 -1988, *Support by an ISDN of Data Terminal Equipment with V-series Type Interfaces with Provision for Statistical Multiplexing*.

[8] CCITT Recommendation X.25 -1984, *Interface between Data Terminal Equipment (DTE) and Data Circuit-Terminating Equipment (DCE) for Terminals Operating in the Packet Mode and Connected to Public Data Networks by Dedicated Circuit*.

D.3 ISO Documents

[9] ISO 8824:1987(E), *Information processing systems — Open Systems Interconnection — Specification of Abstract Syntax Notation One (ASN.1)*.

[10] ISO 8825:1987(E), *Information processing systems — Open Systems Interconnection — Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*.

D.4 Other Documents

[11] NIST Special Publication 500-183, *Stable Implementation Agreements for Open Systems Interconnection Protocols*, Version 4, Edition 1, December 1990.

[12] *STREAMS Programmer's Guide, UNIX System V, Release 3.2* — Prentice Hall.